REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)

2. REPORT DATE
25 Sept. 1995

3. REPORT TYPE AND DATES COVERED
Final Technical 5/26/95 - 9/25/95

4. TITLE AND SUBTITLE
Toward Specification Techniques for Pre-screen projection and other Next-generation User

5. FUNDING NUMBERS
G
N00014-95-11-G014

6. AUTHOR(S)
Robert J. K. Jacob

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Tufts University
Grants & Contracts Administration
Packard Hall
Medford, MA 02155

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of Naval Research
495 Summer Street, Room 103
Boston, MA 02210

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
See attached.

14. SUBJECT TERMS
3-D visualization, non-WIMP interface,

15. NUMBER OF PAGES
15

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	unclassified	unclassified	UL

Toward Specification Techniques for Pre-Screen Projection and Other Next-Generation User

Final Technical Report

Report Date: 25 September 1995

Report Period: Final Report, 26 May 1995 to 25 September 1995

Grant Number: N00014-95-1-G014

Principal Investigator: Robert J.K. Jacob

Address: Department of Electrical Engineering and Computer Science, Tufts University, 161 College Avenue, Medford, Mass. 02155

Scientific Officer: James Templeman

Address: Code 5513, Naval Research Laboratory, Washington, D.C. 20375

Toward Specification Techniques for Pre-Screen Projection and Other Next-Generation User Interfaces

Robert J.K. Jacob

Introduction

We have investigated new languages for describing and implementing next-generation interfaces that might be used in 3-D visualization environments. Working with researchers at NRL Code 5513, we learned about the interfaces they are currently developing and are planning for future work. For current work we examined pre-screen projection and foot control; and we discussed future plans for research in 3-D manipulation, such as 3-D route planning and object-based 3-D visualization. We then studied how new specification languages might capture these types of interfaces and described an approach and conceptual model for doing this. We began with a model and language that combines discrete and continuous interactions, which is described in detail in this report. In collaboration with NRL researchers, we considered its applicability to NRL's 3-D user interface designs. We also began exploring the issue of how to connect the user interface component to a simulation component of a 3-D visualization system.

Basic Model

The first task was to identify the basic structure of next-generation or non-WIMP interaction [4] as the user sees it. The key question is: What is the essence of the *sequence* of interactions in such an interface? Our hypothesis is that it is *a set of continuous relationships, most of which are temporary.*

For example, in a virtual environment, a user might grasp, move, and release an object. The hand position and object position are thus related by a continuous function (say, an identity mapping between the two 3-D positions)—but only while the user is grasping the object. Similarly, using a scrollbar in a conventional graphical user interface, the *y* coordinate of the mouse and the region of the file being displayed are related by a continuous function (a linear scaling function, from 1-D to 1-D), but only while the mouse button is held down. The continuous relationship ceases when the user releases the mouse button.

Some continuous relationships are permanent. In a conventional physical control panel, the rotational position of each knob is permanently connected to some variable. In a flight simulator, the position of the throttle lever and the setting of the throttle parameter are permanently connected by a continuous function.

The essence of these interfaces seems, then, to be a set of continuous relationships some of which are permanent and some of which are engaged and disengaged from time to time. These relationships accept continuous input from the user and typically produce continuous responses or inputs to the system. The actions that engage or disengage them are typically discrete inputs from the user (pressing a mouse button over a widget, grasping an object).

Toward a Model

Most current specification models are based on tokens or events. Their top-down, triggered quality makes them easy to program (and, in fact, everything in a typical digital computer ultimately gets translated into something with those properties). But we see in the above examples that events are the wrong model for describing some of the interactions we need; they are more straightforwardly described by declarative relationships among continuous variables. Non-WIMP interface styles tend to have more of these kinds of interactions.

Therefore, we need to address the continuous aspect of the interface explicitly in our specification model. Continuous inputs have often been treated by quantizing them into a stream of ''change-value'' or ''motion'' events and then handling them as discrete tokens. Instead we want to describe continuous user interaction as a first-class element of our model. Our approach is based on combining dataflow or constraint-like continuous relationships with token-based event handlers. The key here is providing an elegant language that maps onto a user's view of the dialogue and provides good integration between the two parts. The continuous relationships would be described with a data-flow graph, which connects continuous input variables to continuous application (semantic) data and, ultimately, to continuous outputs, through a network of functions and intermediate variables. The result resembles a plugboard or wiring diagram or a set of one-way constraints. Such a model also supports parallel interaction implicitly, because it is simply a declarative specification of a set of relationships that are in principle maintained simultaneously. (Maintaining them all on a single processor within required time constraints is an important issue for the implementation, but should not appear at this level of the specification.)

Note that trying to describe these interfaces in purely continuous terms or purely discrete terms is entirely possible, but silly. For example:

- In the extreme, all physical actions can be viewed as continuous, but we quantize them in order to obtain discrete inputs. For example, the pressing of a keyboard key is a continuous action in space. We quantize it into two states (up and down), but there is a continuum of underlying states, we have simply grouped them so that those above some point are considered ''up'' and those below, ''down.'' We could thus view a keyboard interface in continuous terms. However, we claim that the user model of keyboard input is as a discrete operation; the user thinks of pressing a key or not pressing it.

- Similarly, continuous actions could be viewed as discrete. All continuous inputs must ultimately be quantized in order to pass them to a digital computer. The dragging of a mouse is transmitted to the computer as a sequence of discrete moves over discrete pixel positions and, in typical window systems, processed as a sequence of individual discrete events. However, again, we claim that the user model of such input is as a smooth, continuous action; the user does not think of generating individual ''motion'' events, but rather of making a continuous gesture.

This approach leads to a two-part model of user interaction. One part is a graph of functional relationships among continuous variables. Only a few of these relationships are typically active at one moment. The other part is a set of discrete event handlers. These event handlers can, among other actions, cause specific continuous relationships to be activated or deactivated. A key issue is how the continuous and discrete domains are connected, since a modern user interface will typically use both. One important connection is the way in which discrete events can activate or deactivate the continuous relationships. Purely discrete controls (such as pushbuttons, toggle switches, menu picks) also fit into this framework. They are described by traditional discrete techniques, such as state diagrams and are covered by the ''discrete'' part of this model. That part serves both to engage and disengage the continuous relationships and to handle the truly discrete interactions.

The key to this approach is the hypothesis that the fine-grained aspects of non-WIMP interaction consist of the interplay between continuous and discrete interactions. The two spheres operate relatively independently with communication paths between them. Our initial approach, then, is a model for *combining* data-flow or constraint-like continuous relationships and token-based event handlers. Its goal is to provide a language that integrates the two components and maps closely to the user's view of the fine-grained interaction in a non-WIMP interface. The model thus comprises:

- A set of continuous user interface Variables, some of which are directly connected to input devices, some to outputs, some to application semantics. Some variables are also used for communication within the user interface model (but possibly between the continuous and discrete components), and, finally, some variables are simply interior nodes of the graph containing intermediate results.

- A set of Links, which contain functions that map from continuous variables to other continuous variables. A link may be operative at all times or may be associated with a Condition, which allows it to be turned on and off in response to other user inputs. This ability to enable and disable portions of the data flow graph in response to user inputs is a key feature of the model.

- A set of EventHandlers, which respond to discrete input events. The responses may include producing outputs, setting syntactic-level variables, making procedure calls to the application semantics, and setting or clearing the Conditions, which are used to enable and disable groups of Links.

The model provides for communication between its discrete (event handlers) and continuous (links and variables) portions in several ways:

- Communication from discrete to continuous occurs through the setting and clearing of Conditions, which effectively re-wire the data-flow graph.
- In some situations, there are analogue data coming in, being processed, recognized, then turned into a discrete event. This is handled by a communication path from continuous to discrete by allowing a link to generate tokens which are then processed by the event handlers. A link function might generate a token in response to one of its input variables crossing a threshold. Or it might generate a token when some complex function of its inputs becomes true. For example, if the inputs were all the parameters of the user's fingers, a link function might attempt to recognize a particular hand posture and fire a token when it was recognized.
- Finally, as with augmented transition networks and other similar schemes, we provide the ability for continuous and discrete components to set and test arbitrary user interface variables, which are accessible to both components.

Languages

We are investigating several alternate syntaxes for languages based on this model. The first is a two-part graphical language, in which the continuous and discrete components are described separately. The second is a graphical language that combines the two components. The third is a text-based language.

To illustrate syntax here, we will use a familiar example from a WIMP interface, a simplified slider widget. In it, if the user presses the mouse button down on the slider handle, the slider will begin following the *y* coordinate of the mouse, scaled appropriately. It will follow the mouse continuously, truncated to lie within the vertical range of the slider area, directly setting its associated semantic-level application variable as it moves.

We would view this as a functional relationship between the *y* coordinate of the mouse and the position of the slider handle, two continuous variables (disregarding their ultimate realizations in pixel units). This relationship is temporary, however; it is only enabled while the user is dragging the slider with the mouse button down. Therefore, we provide event handlers to process the button-down and button-up events that initiate and terminate the relationship. Those events execute commands that enable and disable the continuous relationship.

Two-Part Graphical Syntax

The first example, in Figure 1, shows the specification of this simple slider in the two-part graphical notation, with the upper portion of the screen showing the continuous portion of the specification, using solid grey ovals to represent variables, solid grey rectangles for links, and grey arrows for data flows. The lower portion shows the event handler in the form of a state diagram, with states represented as circles and transitions as arrows. This example language illustrates the use of separate continuous and discrete specifications and the way in which enabling and disabling of the continuous relationships provides the connection between the two. Abowd [1] and Carr [2, 3] also present specifications of sliders which separate their continuous and discrete aspects.

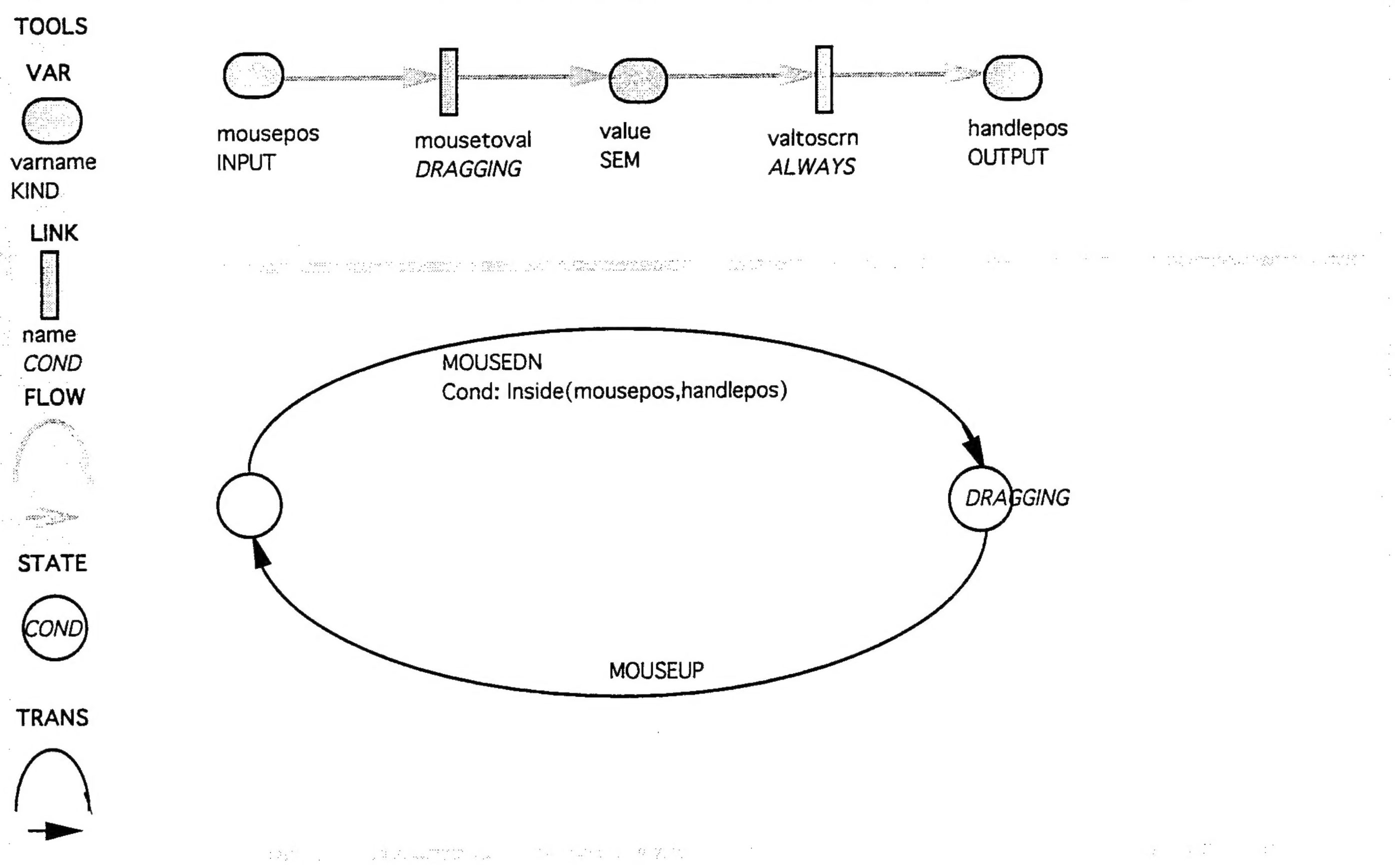

Figure 1. Example of two-part graphical syntax

The continuous relationship is divided into two parts. The relationship between the mouse position and the **value** variable in the application semantics is temporary, while dragging; the relationship between **value** and the displayed slider handle is permanent. Because **value** is a variable shared with the semantic level of the system, it could also be changed by the application or by function keys or other input, and the slider handle would respond. The variable **mousepos** is an input variable, which always gives the current position of the mouse; **handlepos** is an output variable, which controls the current position of the slider handle. The underlying user interface management system keeps the **mousepos** variable updated based on mouse inputs and the position of the slider handle updated based on changes in **handlepos**. The link **mousetoval** contains a simple scaling and truncating function that relates the mouse position to the value of the controlled variable; it is associated with the condition name **dragging**, so that it can be enabled and disabled by the state transition diagram. The link **valtoscrn** scales the variable **value** back to the screen position of the slider handle; it is always enabled.

The discrete portion of this specification is given in the form of a state transition diagram, although any form of event handler specification may be used interchangeably in this system. It accepts a **MOUSEDN** token that occurs within the slider handle and makes a transition to a new state, in which the **dragging** condition is enabled. As long as the state diagram remains in this state, the **mousetoval** link is enabled, and the mouse is connected to the slider handle, without the need for any further explicit specification. The **MOUSEUP** token will then trigger a transition to the initial state, causing the **dragging** condition to be disabled and hence the **mousetoval** relationship to cease being enforced automatically. (The condition names like **dragging** provide a useful layer of indirection where a single condition controls a set of links; in this example there is only one link, **mousetoval**.)

Integrated Graphical Syntax

We can also modify the hybrid approach to unify it into a single representation in which a separate data flow graph is associated with each state, as seen in Figure 2. The idea behind this is to imagine that each state in the state transition diagram has an entire data-flow graph associated with it. When the system enters a state, it begins executing that data-flow graph and continues until it reaches another state. The state diagram can be viewed as a set of transitions between whole data-flow graphs, which can provide a particularly apt description of moded continuous operations (such as grab, drag, and release).

One obvious drawback of this graphical syntax is that it is more difficult to scale it to fit a more complex interface than that of Figure 2 into a single page An interactive editor with rapid zooming would solve this problem.

Text-Based Syntax

The same model lends itself to a text-based language, as shown in Figure 3. Here, we treat each link in the dataflow graph as a separate statement, much like a single rule in a production system. In addition to the inputs, outputs, and computation for each link, we give its enabling condition, that is, the state(s) in which this link will

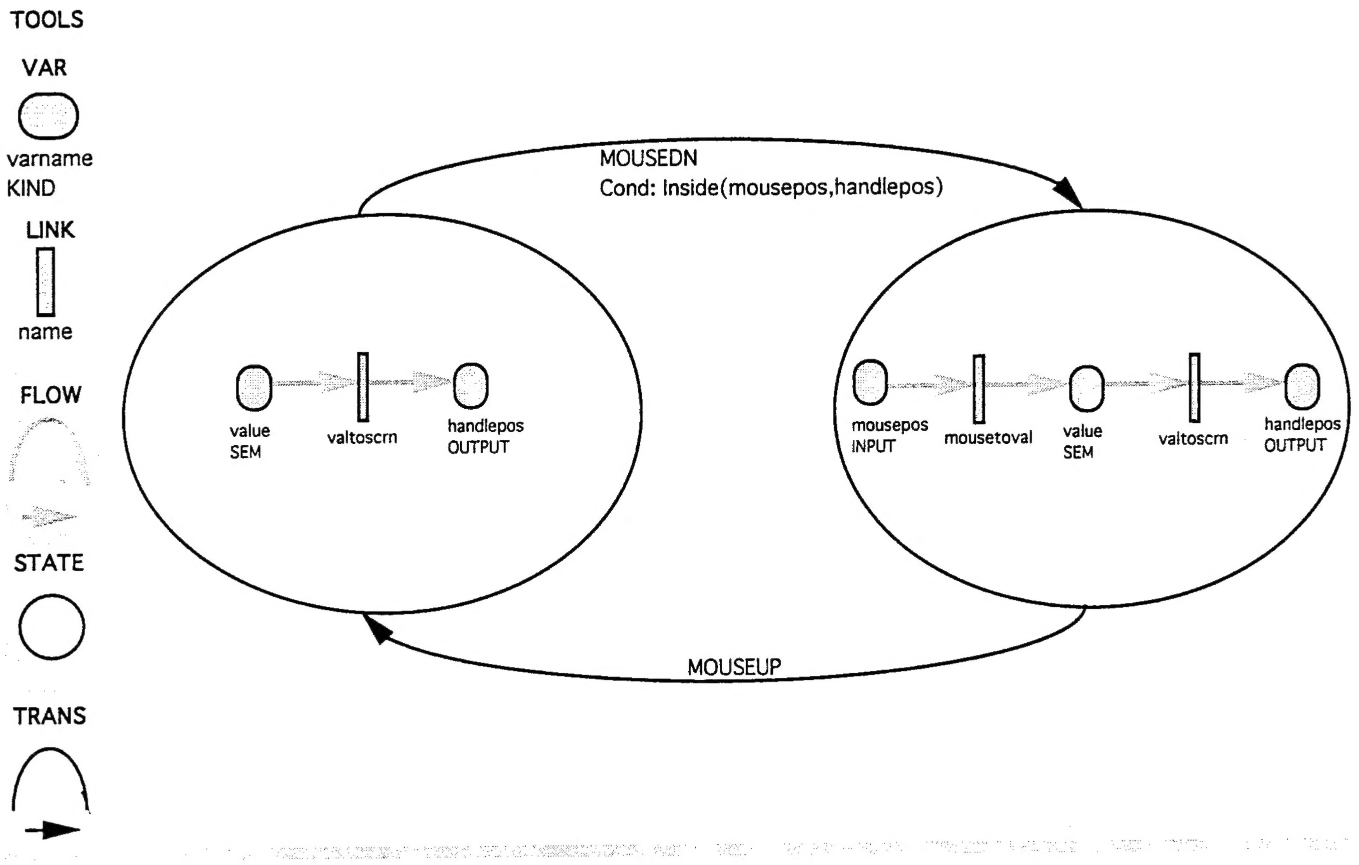

Figure 2. Example of integrated graphical syntax

be active. The remaining step is to define the states themselves; this could be done with the same type of state transition diagram used above, but expressed in textual form as a list of transitions [8].

```
<link>
    <state> /* Empty means this link enabled in all states */
    <in> value
    <out> handlepos
    /* valtoscrn scaling formula goes here */
<link>
    <state> dragging /* i.e., enabled only in state "dragging" */
    <in> mousepos
    <out> value
    /* mousetoval scaling formula goes here */
<std>
    start:     LEFTDN Cond: Inside(mousepos,handlepos); → dragging
```

```
dragging:  LEFTUP →start
```

Figure 3. Example of text-based syntax

NRL Examples

This simple WIMP example provides a concrete illustration of what a specification language might look like. We have shown alternative languages based on our model, and we believe they are a good fit typical 3-D visualization or virtual environment interactions. To begin to test this, we examined some current and future NRL research in this area and considered how the model fits the types of new interaction techniques being developed and planned at NRL Code 5513 and, below, give outlines how they would fit this model. The sketches below suggest, in general terms, how these interaction techniques would fit our model. While they omit details, in each case, the basic fit seems like a fairly natural and straightforward match to the way a user might view the interaction technique.

Typical Navigation in a Virtual Environment

- States: NEUTRAL, FLYING
- State transitions: Make pointing gesture to enter FLYING state, release gesture to return to NEUTRAL state
- Data flow in NEUTRAL state: none
- Data flow in FLYING state: Camera moves forward at constant speed in direction head (or in some systems, hand) is pointing

Pre-Screen Projection [6, 12]

- States: None. This description ignores clutching; clutching would add a second state in which the data flow below is inactive, plus some actions on the state transitions
- Data flow: At all times, head position and orientation continuously control camera position and orientation
- Note: Progressive disclosure may be convenient to represent by adding decision nodes to a data flow graph

Foot Control for Navigation

- States: none (ignoring clutching feature that might be added)
- Data flow: At all times, pressure on foot sensors continuously controls speed and direction of motion
- Note: Data flow graph might be appropriate to represent the relationship between the sensor values and the actual direction of motion
- Note: This assumes a physical model in which speed is a free parameter than can be set directly by the user interface layer (see discussion of layers below).

3-D Route Planning

- Scenario: User's left (or non-dominant) hand can grab and move the scene. Right hand lays down a piecewise linear line through it, using a button press to terminate each segment. Right hand can also edit previous segments.
- Note: The description contains two orthogonal parts and uses substates, much like those used by Harel [5]. The left hand operation is described by two states and a data flow associated with one of them. The right hand operation is described by two states, plus two substates within one of them, plus associated data flows.
- States (left hand): NEUTRAL, MOVESCENE
- State transition (left hand): Make grabbing gesture to enter MOVESCENE state, and set **grabloc** := location of grab at time of transition
- State transition (left hand): Release grabbing gesture to return to NEUTRAL state
- Data flow in MOVESCENE state: Left hand position and orientation continuously controls position and orientation of point **grabloc** within scene
- States (right hand, orthogonal to above): DRAWING, EDITING
- State transition (right hand): Major transition between DRAWING and EDITING states could be function key or other command
- State transition (right hand): In state DRAWING, button click makes transition back to same state, draws a permanent line from **startpos** to right hand, and sets **startpos** := current position of right hand
- Data flow in state DRAWING: Rubber band line is continuously displayed between **startpos** and position of right hand.
- Substates (right hand, within EDITING state): NEUTRAL, MOVEPOINT
- Substate transition (right hand, within EDITING state): Right hand grab gesture located near a line segment endpoint makes transition from NEUTRAL to MOVEPOINT state and sets **p** := that endpoint

- Substate transition (right hand, within EDITING state): Right hand releasing grab gesture makes transition from MOVEPOINT to NEUTRAL, and selected point **p** and the (one or two) line segments attached to it are moved to current position of right hand
- Data flow in state MOVEPOINT (within EDITING state): Selected point **p** and the (one or two) line segments attached to it continuously follow position of right hand
- Note: Initializations are not shown (e.g., **startpos** needs an initial null value and a way to handle that null case)

Testbed

To test this approach, we have begun developing a testbed to support this model. The testbed will be based on a suite of hardware and software that is available at both Tufts and NRL, so our results can continue to be shared in future work. The hardware is an SGI Indigo 2 graphics workstation equipped with 3-D trackers and stereo shutter glasses. Our software will use SGI Performer software for displaying output, and we have developed a constraint algorithm based on the Eval/Vite system from Scott Hudson at Georgia Tech [7] for rapid, real-time solution of the dataflow graphs. We may also develop the constraint algorithm further, to handle time-dependent motions, physical properties, and constraints that refer to other constraints. The state diagrams are executed by an interpreter originally developed at NRL [8, 9]. All software is written in C++ for Unix.

Simulation Component

The next step is to define the boundary between the user interface software and the simulation component for 3-D environments. We began exploring the issue of how to describe, incorporate, or communicate with the simulation component of the system. This is a new issue for 3-D interfaces, because simulation is a very small component of most 2-D interfaces, but a growing portion of 3-D ones. 3-D interfaces often include extensive geometric and physical models and simulations, in contrast to typical 2-D ones. This includes defining the extent of the physical model and how it connects to the rest of the user interface components. While these aspects are, logically, part of the ''user experience,'' we hypothesize that they are a sufficiently separate and extensive realm that they should be treated in separate software layers.

First, we make a separation between the basic physical model or simulation component and the rest of the user interface. Then, we began investigating how the simulation component (1) ought to be expressed and (2) ought to be interfaced to the other user interface components. As a starting point, we posit the simulation as a black box with a set of parameters that can be controlled by the user interface. This leads to a layered model that treats different aspects of the user interface in different layers. As with most such models, its goal is separation of concerns and encapsulation. We want to be able to describe all of the user-visible aspects of the environment, but not to have to think about them all at the same time. The key is designing the right boundaries between the packages.

For many aspects of 3-D environments, the system behavior and corresponding interaction description is, essentially, ''It is just like the physical world.'' That is, the goal of the system design is to create an illusion of objects and behaviors that operate just as they do in the real world. For this, physical modeling is key. While it may be very difficult to achieve and require complex algorithms and clever system design, this behavior is easy to describe simply by describing the real-world counterparts of the objects we are building. Analogously, rendering, too, is complex and tricky, but modern 3-D graphics packages deal with it as a high-level abstraction, simply by giving the camera position and lighting model.

We thus ask what types of interactions should be programmed explicitly in the user interface description language vs. what types can be treated simply as side effects of the operation of a physical model? If bumping into a wall produces sound or haptic feedback, it may be appropriate simply to describe that as part of the physical model and not explicitly as a user interaction. However, if bumping into the wall causes the user to teleport to a new location, this should be described explicitly as a user interaction, since it does not obey a conventional physical model of the world. The analogy in WIMP interfaces is that the action of the cursor following the mouse is usually not thought of as an interface element that is programmed in the UIDL, but simply a permanent part of the hardware and software environment. In 3-D, the analogy applies, but its base physical environment is far, far more complex.

For many early virtual environments this physical model was the entire user interface. However, most systems will ultimately require some capabilities that do not mimic the real world and cannot be described simply by their physical properties. There will be ways to fly or teleport, to issue commands, create and delete objects, search and navigate, or other facilities that go beyond the real-world analogy. These behaviors are of particular interest here, since they are difficult to describe by relying only on the real-world analogy.

A further separation may be made between *geometric* properties of the simulation and other physical properties. Handling the geometric properties, particularly rendering and collision detection, is relatively well understood in 3-D interfaces, and it can be described in fairly standard ways. In contrast, other physical properties have thus far been handled in ad-hoc ways. This leads to the following layered model:

- *Geometric layer:* Behaviors of objects that can be explained entirely by their location and geometry. The software will draw most aspects of the 3-D world simply by traversing the information in this layer and rendering it. Like a display list graphics system, the programmer need not explicitly write code to draw the objects, but needs only to define them and add them to the display list; the underlying software will render them when needed.

- *Physical layer:* Behaviors that can be explained by physical properties other than geometry, but which behave just as they do in the real physical world. They will make use of the geometric information and add other properties and behaviors in this layer.
- *Interface layer:* Behaviors that do not match the physical world, that is, the special commands and operations that have been designed as the user interface. They can manipulate free parameters of objects in the Physical layer (e.g., modify an object) or Geometric layer (e.g., move an object) as well as in the application layer (e.g., create or delete an object).
- *Application layer:* This contains the ''semantics'' or application portion of the system, as with other types of interactive systems. For many early examples of virtual environments, this layer was almost empty.

The boundary between the Physical and Geometric layers is really determined by the state of the art in 3-D graphics software. While systems for handling and rendering geometry are well established, systems for handling other aspects of physical simulation are much less standard. Defining this boundary allows us to take advantage of the standard capabilities. Its real purpose is to separate aspects of the simulation that are well understood and packaged from those that must still be programmed in ad-hoc or application specific ways. Currently, only geometry falls into the first category, but, in the future, we hope to take advantage of progress in 3-D physical simulation software, such as that currently being sponsored by ONR (SHASTRA and Isaac projects at Purdue, Baraff's work at CMU). As such work becomes established, the boundary between these two layers may move or become unnecessary.

The boundary between the Physical and Interaction layers is determined approximately by asking whether a behavior can be described simply by saying that it is supposed to operate exactly like the real world. Observe that if all interaction within the 3-D environment were precisely modeled on corresponding physical actions, the Interaction layer would be empty. However, much of the benefit of realistic computer-generated environments is their ability to mimic the real world *plus* provide additional capabilities only available in a computer.

It may also turn out to be useful to define additional, higher layers in this model. Some VR applications are beginning to combine the notion of intelligent agents, sometimes in the form of user-visible animated characters in the display. In an agent-based interface, the lower layers would still contain the mechanisms for interacting with the agents and changing their parameters. However, the behavior of the agents caused by such parameter changes, or the ''goals'' or ''personalities'' of the agents may be specified separately, in a higher level. Similarly, in an interface involving discourse-based interaction, or other higher-level constructs, their description would come at a higher ''discourse'' level, which might be separated into its own module [10, 11].

Future Work

The next step in this research will be to test and stabilize the language and express some of the above specifications formally. At the same time, we will continue building the testbed. Then, we will see how well the new 3-D interaction techniques can be implemented on our testbed, directly from their descriptions. Finally, we will refine the simulation component and its interface to other software components.

References

1. G.D. Abowd and A.J. Dix, ''Integrating Status and Event Phenomena in Formal Specifications of Interactive Systems,'' *Proc. ACM SIGSOFT'94 Symposium on Foundations of Software Engineering*, Addison-Wesley/ACM Press, New Orleans, La., 1994.
2. D. Carr, ''Specification of Interface Interaction Objects,'' *Proc. ACM CHI'94 Human Factors in Computing Systems Conference*, pp. 372-378, Addison-Wesley/ACM Press, 1994.
3. D.A. Carr, N. Jog, H.P. Kumar, M. Teittinen, and C. Ahlberg, ''Using Interaction Object Graphs to Specify and Develop Graphical Widgets,'' Technical Report ISR-TR-94-69, Institute For Systems Research, University of Maryland, 1994.
4. M. Green and R.J.K. Jacob, ''Software Architectures and Metaphors for Non-WIMP User Interfaces,'' *Computer Graphics*, vol. 25, no. 3, pp. 229-235, July 1991.
5. D. Harel, ''On Visual Formalisms,'' *Comm. ACM*, vol. 31, no. 5, pp. 514-530, May 1988.
6. D. Hix, J.N. Templeman, and R.J.K. Jacob, ''Pre-Screen Projection: From Concept to Testing of a New Interaction Technique,'' *Proc. ACM CHI'95 Human Factors in Computing Systems Conference*, pp. 226-233, Addison-Wesley/ACM Press, 1995. http://www.acm.org/sigchi/chi95/Electronic/documnts/papers/dh_bdy.htm [HTML]; http://www.cs.tufts.edu/~jacob/papers/chi95.txt [ASCII].
7. S. Hudson and I. Smith, ''Practical System for Compiling One-Way Constraint into C++ Objects,'' Technical Report, Georgia Tech Graphics, Visualization, and Usability Center, 1994.
8. R.J.K. Jacob, ''An Executable Specification Technique for Describing Human-Computer Interaction,'' in *Advances in Human-Computer Interaction, Vol. 1*, ed. by H.R. Hartson, pp. 211-242, Ablex Publishing Co., Norwood, N.J., 1985.

9. R.J.K. Jacob, ''A Specification Language for Direct Manipulation User Interfaces,'' *ACM Transactions on Graphics*, vol. 5, no. 4, pp. 283-317, 1986. http://www.cs.tufts.edu/~jacob/papers/tog.txt [ASCII]; http://www.cs.tufts.edu/~jacob/papers/tog.ps [Postscript].

10. M.A. Perez and J.L. Sibert, ''Focus in Graphical User Interfaces,'' *Proc. ACM International Workshop on Intelligent User Interfaces*, Addison-Wesley/ACM Press, Orlando, Fla., 1993.

11. M.A. Perez and R.J.K. Jacob, ''A UIMS Architecture for Focus Processing in a Graphical User Interface,'' *AAAI Symposium on Intelligent Multi-Media Multi-Modal Systems*, pp. 87-92, AAAI, Stanford, Calif., 1994.

12. J. Templeman, ''Pre-screen Projection of Virtual Scenes,'' *Proc. Virtual Reality Systems Conference*, SIG Advanced Applications, New York, N.Y., 1993.